

# LOW-RANK APPROXIMATE SINGULAR VALUE DECOMPOSITION AND IMAGE COMPRESSION

LUCAS DE LA BROSSE, ANURAG MIGLANI, THOMAS ZAMOJSKI



École nationale de la statistique  
et de l'analyse de l'information

## 1. INTRODUCTION

We chose in our project to investigate an aspect of Big Data that is probably different from the other groups, that has been heard about many times in the program, but has seen few instances of it so far. Our data might be voluminous, but not necessarily. The problem rather stems from the need to process that data in a certain way that requires too much computational power, even for a data volume of the order of 1GB. Moreover, the data come in rapidly and must be processed with small latency. To address this Big Data issue, we discuss a compressive sensing method via probabilistic low-rank approximations.

We will describe how these problems occur in scientific imaging and make some empirical tests on images. However, we will try our best to focus on the aspects of interest to all data scientists. In particular, we will describe the tools we used having in mind that they should be useful in many different contexts, and we hope that people will find at least something of interest to them.

**1.1. Application in Imaging.** Hyperspectral Imaging (HSI) collects and processes information from an as wide as possible spectrum of wavelengths, going far beyond the visible spectrum. If the human eye sees visible light in mostly three bands (red, green and blue), one can divide the whole spectrum of electromagnetic radiations' frequency into several hundreds to thousands of fine bands. Moreover, a single area is sampled many times for image correction to account for a variety of measurement errors. A single hyperspectral image can therefore be quite big in itself, 2-3 Gigabytes being not rare.

Transmission, storage and processing of such large sets of data is practically challenging. Spectrometers are mounted on aircrafts or satellites, neither of which has the appropriate computing power or storage to deal with HSI. Dimensionality reduction methods provide solutions to those difficulties. The airborne device encodes the images via compressive sensing methods and send it to the ground where more performant computers are available for decoding and all the heavy lifting.

A recurrent paradigm in these compressive sensing methods is to represent images as matrices, compress them to lower-dimensional matrices, which are then factorized. The result is a low-rank matrix approximation to the original image matrix.

**1.2. Probabilistic Methods.** Probabilistic methods to deal with hard problems have a long history, and it comes as no surprise that they provide powerful computational solutions in the world of Big Data. In 2006, Martinsson, Rokhlin and Tygert published a new approach to probabilistic low-rank matrix approximations that we will refer to as randomized singular value decomposition (rSVD) [4]. A survey and extension on the subject can be found in the highly recommended seminal paper of Halko, Martinsson and Tropp [5]. Furthermore, Martinsson and Voronin provide an implementation of the

algorithm with its description in [3], while Candès and Witten found a novel and intuitive analysis of the algorithm in [1]. Finally, rSVD was applied to HSI in [2], where it was also compared to another probabilistic method, compressive-projection principal component analysis (CPPCA).

Statisticians, perhaps not so concerned with HSI, should nonetheless be interested in low-rank matrix approximation and rSVD. For example, it provides a performant solution to large principal component analysis (PCA).

**1.3. Pragmatic Goals of the Project.** Very popular amongst statisticians, the R language has many pros and cons. One of the cons we have encountered is that it is not designed for numerical linear algebra. Nonetheless, we would like to bring a practical solution in R for doing rSVD and programming numerical linear algebra in general.

The project thus consists in first installing a framework for efficient numerical linear algebra in R capable of working with large matrices and allowing for parallelisation as much as possible. The framework should bridge the gap between R and C++. We then implement rSVD in this framework and test its performance in the setting of image compressive sensing. Although HSI brings interesting challenges, it is mostly outside the scope of the current project. We rather restrict our focus to the usual color images in the visible spectrum, as it is easier for a wider audience to visualise, including ourselves.

TODO: include description of dataset.

Prior to this project, we had no knowledge about the topics covered. We would therefore appreciate receiving any constructive comments about it.

## 2. THE FRAMEWORK

Our ambitious goal is to create a setup around the R language and C++ not only to implement rSVD, but that could be truly useful for us and other data scientists in the future. The framework should be easy to use, provide efficient functionalities in numerical linear algebra, use efficiently memory, be out-of-core friendly, and finally allows for parallelization as much as possible. Fortunately, the tools have already been created by the community.

**2.1. Bigmemory R Package.** The following brief discussion is mostly taken from (ref to bigmemory vignette), to which we refer for a more thorough introduction to bigmemory.

**2.1.1. Objectives of bigmemory.** Data frames and matrices in R were designed for easy and fast manipulation on datasets that are much smaller than available RAM. A second category of datasets arise when they require more than available RAM. In that case, many R packages exist to manipulate such out-of-core data. However, the disadvantage is that the user is forced to wait for disk accesses, and they are not well-suited for synchronization between multiple processes. The bigmemory package wants to address a third kind of datasets: massive datasets of the order of 1GB that although voluminous, still fit into memory.

Why do we need this third category? R makes excellent choices in design that however have noticeable side-effects for memory when it is becoming rare. Try the following 2.24GB and 1.12GB memory consumption examples:

```
R> x <- matrix(0, 1e+08, 3); round(object.size(x) / (1024)^3, 2)
[1] 2.24
R> x <- matrix(as.integer(0), 1e+08, 3); round(object.size(x) / (1024)^3, 2)
[1] 1.12
```

What happens here is that R has 8-byte real numbers versus 4-byte integers, and numerical values are considered real by default. Ok, so suppose you are careful and use the second alternative, and then manipulate the matrix some more later:

```
R> x <- matrix(as.integer(0), 1e+08, 3)
R> x <- x + 1
```

What is the memory usage? No, it is not 1.12GB. Not even 2.24GB, but rather 3.36GB! There is 1.12GB for the original matrix, which is then copied and coerced to real numbers for another 2.24GB, getting a peak of 3.36GB before releasing the original matrix to get down to 2.24GB.

Other similar examples can be constructed, some even unknown to the authors. It is important however to realise that one can get easily trapped by these side-effects when in the third category of datasets, those that are voluminous but still fit in memory, at least until you manipulate them.

This is where the package `bigmemory` and its `big.matrix` class create memory efficiencies. In addition, they also create opportunities for parallel computing, which is becoming increasingly important as computing power is slowly caping-off.

**2.1.2. Using `bigmemory`.** The `big.matrix` class plays the central role in the package. It works similarly to usual matrices, for example `x[,2]` gives back the second column. However, it is a pointer to the actual matrix in memory and thus is passed by reference to functions. Some usual R functions are implemented for `big.matrix` like `nrow()`, `ncol()`, and `summarise()`, some in a more efficient way like `mwhich()`. There are several ways to initialise a `big.matrix`. One can use the constructor `big.matrix()` or typecast a matrix with `as.big.matrix()`. However, since our matrices are big, we will rather read them in from a file with a `big.matrix` version of `read.table()`:

```
redMat <- read.big.matrix(filename = "data/smallRed.txt", sep = ' ', type = "short",
                          backingpath = "./data/", backingfile = "smallRed.bin",
                          descriptorfile = "smallRed.desc")
```

The version we used here creates a backedup `big.matrix` on the filesystem. The `type` argument refers to the `c++` type used to write the entries and can be any of `short`, `integer`, `double`. One of the current most problematic restriction is that `big.matrix` supports only atomic types, that is the entries must be all of the same type. It is therefore the user's duty to make sure that matrices are converted to a single type.

The backing file is specified with the `backingpath` and `backingfile` arguments. The binary file it creates is simply the raw bytes of the entries written one after the other, column by column. This is referred in matrix jargon as column-dominant serialization. This makes it easy to read in memory using our own method in `C++`.

The `descriptorfile` stores the S4 object returned by the R `describe()` function. It has one attribute called `description`, which is a list of variables such as `nrow`, `ncol`, `offsets`, `filename`, etc... However, it does not store the `backingpath`, probably because that should be machine specific.

The first time one invokes `read.big.matrix` in R takes some non-negligible amount of time to create the files. However, if one backed it up on the filesystem as above, then you don't need to read it in again. Instead, you just attach it to your session:

```
redMat <- attach.big.matrix("data/smallRed.desc")
```

There are sister packages to `bigmemory` that implements some functionalities for `big.matrix`. For example, `biganalytics` implements `bigglm` to do efficient generalised linear regression on a `big.matrix`. Another one is `bigalgebra`, which was supposed to implement linear algebra manipulation. However, it has only the arithmetic operations and has been under "heavy development", although the last update on github was in 2014. We therefore do not use `bigalgebra` at all.

**2.2. C++ Linear Algebra Libraries.** `C++` has many efficient libraries for numerical analysis that are commonly used by developers. In fact, R and octave themselves also rely on these libraries for some functionalities. For example, QR-decomposition in R is done through LINPACK or its more recent fork LAPACK. Naturally, we would like to make use of this power as well. We will therefore install `openBLAS` and `Armadillo`. It is better to do it in that order, as `Armadillo` needs to find `openBLAS` to configure itself to work with it.

**2.2.1. `openBLAS`.** BLAS stands for basic linear algebra subprograms. These are routines that provide standard building blocks for performing basic vector and matrix operations. BLAS are efficient, portable and widely available. Higher linear algebra softwares like LAPACK use BLAS.

However, we want to make use of our multicore architecture. `OpenBLAS` is an open source implementation of BLAS API with many optimizations for specific processor types and is developed at the Lab of Parallel Software and Computational Science ISCAS. It claims to achieve similar performance as the proprietary freeware Intel MKL (math kernel library). We think therefore it is a good choice.

Quick installation is quite easy:

```
cd ~/pkg
git clone https://github.com/xianyi/OpenBLAS
cd OpenBLAS
make FC=gfortran
sudo make PREFIX=/opt/openblas install
```

This makes an installation in the `/opt` folder as to not collide with `apt-get`. OpenBLAS is also available through `apt-get`'s `libopenblas-dev` package for Debian/Ubuntu.

**2.2.2. Armadillo.** The efficient C++ library *Armadillo* is open source software that provides an easy to use API similar to matlab, integrates with other great and common libraries such as LAPACK, BLAS, openBLAS, ATLAS, and Intel MKL and integrates seamlessly with R via Rcpp and RcppArmadillo. Moreover, it employs delayed evaluation and optimisation through template metaprogramming. This is why we chose to use Armadillo for matrix computations.

For installation on Linux, first make sure LAPACK, BLAS and BOOST are already installed on your computer before installing Armadillo:

```
sudo apt-get install liblapack-dev libblas-dev libboost-dev
sudo apt-get install libarmadillo-dev
```

Normally, configurations should be automatically done, but if need be, some config files are to be found in

```
nano /usr/include/armadillo_bits/config.hpp
```

We can work with Armadillo directly with R. Rcpp is a R package for seamless C++ integration. It uses C++ objects to mimic R's objects, allowing for easier argument passing from one language to another. On top of Rcpp, RcppArmadillo adds the integration with Armadillo's objects such as vectors and matrices.

Armadillo is surprisingly well-documented and has many adepts on forums. We refer to the official website for documentation: <http://arma.sourceforge.net/docs.html>. Our code will provide further examples, and we will explain it as we go.

### 3. RANDOMIZED LOW-RANK SINGULAR VALUE DECOMPOSITION

In this section, we will briefly describe the rSVD algorithm.

Given a  $m \times n$  matrix  $A$ , the singular value decomposition (SVD) finds a  $m \times m$  orthogonal matrix  $U$ , a  $n \times n$  orthogonal matrix  $V$  and a nonnegative diagonal  $m \times n$  matrix  $D$  such that  $A = UDV^T$ . The columns  $u_i$  of  $U$  are the left singular vectors, the columns  $v_i$  of  $V$  are the right singular vectors and the diagonal elements  $\sigma_i$  of  $D$  are the singular values.

Given a number  $k$  smaller than both the number of rows and the number of columns of  $A$ , we can truncate  $D$  by keeping  $\sigma_1$  through  $\sigma_k$  and setting the other singular values to zero. We denote such truncated matrix by  $D_k$ . Then  $A_k = UD_kV^T$  is the best approximation to  $A$  of rank  $k$  in the sense that it minimizes mean square error.

Computing SVD is well-optimized. Nonetheless, it requires quite a lot of computation power for large matrices and can be infeasible when time is limited. rSVD provides a very fast randomized low-rank approximation similar to truncated SVD.

- **Input:** Large  $m \times n$  matrix  $A$ , the rank  $k$ , integral oversampling parameter  $p$  (usually 3 to 10 works well).
- **Output:**  $k$  approximate left singular vectors in  $U$ , right singular vectors in  $V$  and singular values in  $D$ .

- (1)  $W \leftarrow$  random  $n \times (k + p)$  matrix. Can be sampled as gaussian or uniform for example.
- (2)  $Y \leftarrow AW$ . Needs one pass through the large matrix  $A$ .  $Y$  is a random sample of the image of  $A$ .
- (3)  $Q \leftarrow$  orthogonalisation of  $Y$ . This can be done via economical QR-decomposition.
- (4)  $B \leftarrow Q^T A$ . Needs another pass through the matrix  $A$ .
- (5) Compute  $B^T = \hat{Q}R$  via economical QR-decomposition.
- (6) Compute  $R = \hat{U}\hat{D}\hat{V}^T$  via SVD. Fast as  $R$  is a  $(k + p)$ -square matrix.
- (7) Assume  $A \approx QQ^T A = QB = Q\hat{V}\hat{D}\hat{U}^T\hat{Q}^T$ , gives the desired decomposition.

Return the truncated  $U = (Q\hat{V})_k$ ,  $V = (\hat{Q}\hat{U})_k$  and  $D = \hat{D}_k$ .

The algorithm oversamples the range of  $A$  for numerical stability. Note that here the rank  $k$  of the approximation is an input, but we could easily implement a rank-revealing version of the algorithm, at the cost of computation speed. Also, the current version makes two passes through the matrix  $A$ . That could be inconvenient for a truly massive matrix stored in dead memory and/or distributed. There is a way to improve the above to make only one pass through  $A$ . We refer the interested reader to [3] for details and other variations of the algorithm.

## 4. IMPLEMENTATIONS IN R

In R, SVD is implemented with the help of LAPACK. It uses our multicore architecture. Yet, it is as mentioned slow for large matrices.

An implementation of rSVD using ordinary matrices has been provided in R through the package named `rsvd`. It includes several versions of the algorithm, and we will compare our implementation with this one as well.

We now give details of our implementation in C++, making the assumption that the matrices are to be found in the data directory at the root of the project:

```
#include <iostream>
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]

using namespace Rcpp;
using namespace arma;
using namespace std;

// Loads big.matrix into an arma::mat.
mat readBigMatrix(string fname, int nr, int nc){...}

// [[Rcpp::export]]
List myrsvd(S4 bigmat, int k, int p){
  mat Qb, R, Q, U, V, tmp;
  vec s;

  List desc = bigmat.slot("description");
  string fname = desc["filename"];
  fname = "data/" + fname;
  int nr = desc["nrow"];
  int nc = desc["ncol"];

  mat A = readBigMatrix(fname, nr, nc);
  mat W = randn(nc,k+p);
  cerr << "Computing the ortho basis for approximate range\n";
  qr_econ(Q,tmp, A * W);
  cerr << "Computing the QR decomposition\n";
  qr_econ(Qb, R, A.t() * Q);
  cerr << "Computing the SVD of low-rank\n";
  svd(U, s, V, R);
  cerr << "Returning the values to R\n";
  mat Uf = Q * V; Uf = Uf.cols(1,k);
  mat Vf = Qb * U; Vf = Vf.cols(1,k);
  s = s.subvec(1,k);
  return List::create(
    Named("u") = Uf,
    Named("d") = s,
    Named("v") = Vf);
}
```

Concerning Armadillo, here we have made use of some of its core features. The types `arma::mat` and `arma::vec` implements vectors and matrices, and are handled similarly to `matlab`. Some random generators are included, such as `randn` above for random gaussian matrices, again similar to `matlab`'s `randn`. Finally, through integration with `openBLAS`, we have made use of matrix decompositions `qr_econ` and `svd`.

The compiler understands annotation, which is very convenient. The first function, `readBigMatrix`, is for the C++ program only. On the other hand, because of the `Rcpp::export` annotation, `myrsvd` will be callable from R.

Another convenient feature is that one can mix R, Rcpp and Armadillo types. For example, a `Rcpp::export` function returning a `arma::mat` will work with R through `RcppArmadillo`'s automatic conversions. In the code above, we used one instance of this conversion with the `int` inputs.

As seen above, one can work in C++ with R's S4 objects. We have made use of this feature to pass a `big.matrix` descriptor to our function in C++, thus playing well with the `bigmemory` package.

To invoke the function in R is now easy:

```
require(bigmemory)
require(Rcpp)
require(RcppArmadillo)

redMat <- attach.big.matrix("data/smallRed.desc")
redDesc <- describe(redMat)

sourceCpp("src/myrsvd.cpp")
ll <- myrsvd(redDesc,500,10)
```

## 5. EMPIRICAL RESULTS

TODO: agree on a dataset, should have access to raw pixel data, like with `pnm`-family. For now, we've been using about 700MB picture of `andromede`.

TODO: Run pure SVD, `rsvd` and `myrsvd`, microbenchmark them, look at memory and cpu consumption, plot all singular values on a single plot, remark that `rSVD` systematically underestimates the values.

TODO: Show the image compression quality at different levels of compression, controlled by rank  $k$ .

MAYBE: If time permits, compare with JPEG.

PROB NOT: If time permits, try to get some HSI images. Not only need to have raw pixels, but also the tools to display those images...

## REFERENCES

- [1] E.CANDÈS, AND R.WITTEN. Randomized algorithms for low-rank matrix factorizations: sharp performance bounds. *Algorithmica* 72, 1 (2015), 264–281.
- [2] J.ERWAY, X.HU, R.PLEMMONS, J.ZHANG, AND Q.ZHANG. Randomized svd methods in hyperspectral imaging. *Journal of Electrical and Computer Engineering 2012* (2012), 15 pages.
- [3] MARTINSSON, P.-G., AND S.VORONIN. Rsvdpack: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core and gpu architectures.
- [4] MARTINSSON, P.-G., AND V. ROKHLIN, M. T. A randomized algorithm for the approximation of matrices. *Computer Science Dept. Tech. Report 1361, Yale Univ., New Haven, CT* (2006).
- [5] N.HALKO, MARTINSSON, P.-G., AND J.A.TROPP. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decomposition. *SIAM review* 53, 2 (2011), 217–288.